LAB 3 - PromQL

Prometheus data model

Tout ce que vous devez savoir est très bien expliqué dans la Prometheus documentation https://prometheus.io/docs/concepts/data_model/

À la base, le modèle de données représente les séries chronologiques sous la forme d'un nom de métrique suivi d'un ensemble de noms et de valeurs d'étiquette. Les échantillons sont des valeurs numériques associées à une time series à un horodatage donné.

Voici la représentation d'une métrique avec différentes étiquettes:

```
process_resident_memory_bytes{instance="localhost:9100",job="node_exporter"}
```

```
process_resident_memory_bytes{instance="localhost:9090",job="prometheus"}
```

Le type d'une métrique peut être l'un des suivants:

- Counter (`node_cpu_seconds_total`)
- Gauge (`node_memory_MemFree_bytes`)
- Histogram (`prometheus_http_request_duration_seconds`)
- Summary (`go_gc_duration_seconds`)

L'histogramme et le Summary sont liés et offrent différents compromis. La principale différence du point de vue de l'utilisateur est que les histogrammes peuvent être agrégés alors que les Summary ne le peuvent pas (en général).

Exercice: visitez à nouveau le point de terminaison des métriques (/metrics) de Prometheus et vérifiez les différents types de métriques.

Prometheus query language (PromQL)

PromQL est un langage puissant qui permet de découper et de regrouper les données selon les besoins.

Une expression peut vous obtenir toutes les time series pour un nom de métrique donné:

```
prometheus_http_requests_total
```

Cela nous donne toutes les requêtes http, mais nous avons 2 problèmes:

1. Il y a trop de points de données pour déchiffrer ce qui se passe.
2. Vous remarquerez que cela **prometheus_http_requests_total** ne fait qu'augmenter, car c'est un compteur. Ceux-ci sont courants dans Prometheus, mais ne sont pas utiles pour représenter graphiquement.

Il Est Facile De Filtrer Par Étiquette.

```
prometheus_http_requests_total{job="prometheus", code="200"}
```

Vous Pouvez Vérifier Une Sous-Chaîne À L'aide De La Correspondance Regex.

```
prometheus_http_requests_total{code=~"2.*"}
```

Si vous souhaitez en savoir plus, voici la documentation sur Regex.

Types de données de PromQL

PromQL utilise quatre types de données

- Scalaire
- Vecteur de portée
- Vecteur instantané
- Chaîne de caractères

Scalaire :

Les expressions résultant en un seul nombre flottant numérique constant sont scalaires.

exemple, 11,99

Chaîne :

Les expressions dont la sortie est un littéral de chaîne font partie de cette catégorie. Il est actuellement inutilisé dans Prometheus.

par exemple, bienvenue

Vecteurs instantanés :

Ces vecteurs ont une valeur unique correspondant à chaque horodatage de la série temporelle. Vous pouvez exécuter l'expression ci-dessous sur votre navigateur d'expressions

```
prometheus_http_requests_total
```

Ainsi, vous pouvez constater qu'il a renvoyé toutes les séries temporelles avec le nom **prometheus_http_request_total** et il a renvoyé une valeur de la série temporelle donnée.

Range Vector

Ces vecteurs ont une liste de valeurs correspondant à chaque horodatage de la série temporelle.

Rappel: seuls les vecteurs instantanés peuvent être représentés graphiquement. Vous pourrez bientôt voir comment visualiser les vecteurs de portée à l'aide de fonctions.

```
prometheus_http_requests_total[5m]
```

Vous pouvez constater que le vecteur de plage peut renvoyer de nombreux échantillons pour chaque série temporelle. Les durées sont spécifiées sous forme de nombre, suivi immédiatement de l'une des unités suivantes :

- ms – millisecondes
- s – secondes
- m – minutes
- h – heures
- d – jours – en supposant qu'un jour a toujours 24h
- w - semaines - en supposant qu'une semaine a toujours 7j
- y – années – en supposant qu'une année a toujours 365j

```
node_cpu_seconds_total
```

Ajoutez un sélecteur d'étiquettes pour filtrer un processeur particulier :

```
node_cpu_seconds_total{cpu="0"}
```

Mais obtenir le nombre total de secondes qu'un processeur a passé dans chaque mode n'est pas vraiment significatif. Et si vous souhaitez obtenir le pourcentage de temps passé? C'est là que nous devons introduire des vecteurs et des fonctions de plage de temps (**`rate ()`** en particulier).

Jusqu'à présent, nous n'avons utilisé que des sélecteurs de vecteurs instantanés qui renvoient un seul échantillon pour chaque time series correspondant aux sélecteurs d'étiquettes donnés.

Avec les sélecteurs de range vectors, Prometheus renvoie la liste des échantillons dans la plage de temps donnée pour chaque série de temps.

```
node_cpu_seconds_total{cpu="0"}[5m]
```

Encore une fois, cette expression n'est pas très utile en elle-même mais c'est là que `rate ()` vient à la sauver:

```
rate(node_cpu_seconds_total{cpu="0"}[5m])
```

Vous remarquerez que nous sommes en mesure de représenter graphiquement toutes ces fonctions. Étant donné que seuls les vecteurs instantanés peuvent être représentés graphiquement, ils prennent un vecteur de plage comme paramètre et renvoient un vecteur instantané.

Augmentation de la moyenne Http_requests_total Sur Les 5 Dernières Minutes.

```
rate(prometheus_http_requests_total[5m])
```

Exercice: écrivez une requête qui renvoie le pourcentage de temps processeur inactif (indice: PromQL prend en charge les opérateurs arithmétiques).

Solution

```
100 * rate(node_cpu_seconds_total{cpu="0",mode="idle"}[5m])
```

irate

Examine les 2 échantillons les plus récents (jusqu'à 5 minutes dans le passé), plutôt qu'une moyenne comme rate

```
irate(http_requests_total[5m])
```

Il est préférable de l'utiliser rate lors des alertes, car cela crée un graphique fluide puisque les données sont moyennées sur une période de temps.

Rate vs irate

La différence réside dans la façon dont le taux est calculé. « taux » prend la différence entre le premier et le dernier point de données dans la fenêtre de temps sélectionnée, et donc lisse sur une période de temps plus longue :

```
{label="Z"}  ...*.....*.....*.....*.....*...     v
                <--------- rate -------->
```

Alors que « irate » (taux instantané) prend la différence entre les deux derniers points de données dans la fenêtre de temps sélectionnée :

```
{label="Z"}  ...*.....*.....*.....*.....*...     v
                                  <----->
                                   irate
```

Requêtes HTTP Dans la dernière Heure.

Ceci est égal au rate * # de secondes

```
increase(prometheus_http_requests_total[1h])
```

Ce ne sont qu'une petite fraction des fonctions, exactement ce que nous avons trouvé le plus populaire. Vous pouvez trouver le reste https://prometheus.io/docs/prometheus/latest/querying/functions/

Pour Les Vecteurs Instantanés

Répartition parle code de statut

```
sum(rate(prometheus_http_requests_total[5m]))
```

Vous remarquerez que **`rate(prometheus_http_requests_total[5m])`** ci-dessus fournit une grande quantité de données. Vous pouvez filtrer ces données à l'aide de vos étiquettes, mais vous pouvez également regarder votre système dans son ensemble en utilisant **sum**(ou faire les deux).

Vous pouvez également utiliser **min**, **max**, **avg**, **count** et de la **quantile** même. Cette requête vous indique le nombre total de requêtes HTTP, mais n'est pas directement utile pour déchiffrer les problèmes de votre système. Je vais vous montrer quelques fonctions qui vous permettent d'avoir un aperçu de votre système.

Exemple : Pour calculer la quantité d'utilisation du processeur par hôte dans votre cluster, nous devons additionner tous les modes à l'exception de idle, iowait, guest et guest_nice.

Le PromQL ressemble à ceci :

```
sum(rate(
        node_cpu_seconds_total{mode!="idle",
```

```
                mode!="iowait",
                mode!~"^(?:guest.*)$"
                }[5m])) BY (instance)
```

Somme par code d'état

```
sum by (code) (rate(prometheus_http_requests_total[5m]))
```

Vous pouvez également utiliser ***without*** plutôt que ***by***

Maintenant, vous pouvez voir la différence entre chaque code d'état.

Décalage

Vous pouvez utiliser ***offset*** pour modifier l'heure des vecteurs instantanés et de portée. Cela peut être utile pour comparer l'utilisation actuelle à l'utilisation passée lors de la détermination des conditions d'une alerte.

```
sum(rate(prometheus_http_requests_total[5m] offset 5m))
```

N'oubliez pas de mettre offset directement après le sélecteur.

Les Opérateurs

Dans PromQL, il existe quatre types de Matchers

- Matcher d'égalité (=)
- Matcher d'égalité négative (!=)
- Regular expression matcher(=~)
- Negative Regular expression matcher(!~)

Matcher d'égalité (=) :

Sélectionnez des étiquettes qui sont exactement égales à la chaîne fournie. Et il est sensible à la casse.

Supposons que vous travailliez sur une cible spécifique telle qu'un exportateur de nœuds et que vous ne vouliez pas voir la métrique provenant d'une tâche cible autre que l'exportateur de nœuds.

Pour filtrer l'exportateur de nœud de travail de série temporelle, la requête comme celle-ci :

par exemple, **process_cpu_seconds_total{ job="Node Exporter" }**

Matcher d'égalité négative (!=) :

Sélectionnez des étiquettes qui ne sont pas égales à la chaîne fournie. En langage simple, cela signifie si vous ne voulez pas voir de séries temporelles spécifiques

par exemple, **process_cpu_seconds_total{ job!="Node Exporter" }**

Regular expression matcher (=~) :

Sélectionnez les étiquettes qui correspondent à l'expression régulière de la chaîne fournie.

par exemple, **prometheus_http_requests_total{ handler=~"/api.*" }**

Negative Regular expression matcher (!~) :

Sélectionnez les étiquettes qui ne correspondent pas à l'expression régulière de la chaîne fournie.

par exemple, **prometheus_http_requests_total{ handler!~"/api.*" }**

Opérateurs Prometheus dans PromQL

- Opérateurs binaires
- Opérateurs d'agrégation

Opérateurs binaires :

Les opérateurs binaires sont les opérateurs qui prennent deux opérandes et effectuent les calculs spécifiés sur eux.

Types d'opérateurs binaires

Il existe trois types d'opérateurs binaires

- Opérateurs binaires arithmétiques
- Opérateurs binaires de comparaison
- Opérateurs binaires logiques/définis

Opérateurs binaires arithmétiques

Les opérateurs arithmétiques sont les symboles qui représentent les opérations mathématiques arithmétiques.

Les opérateurs arithmétiques binaires suivants existent dans Prometheus :

- + (ajout)
- - (soustraction)
- * (multiplication)

- / (division)
- % (module)
- ^ (puissance/exponentiation)

Les opérateurs arithmétiques binaires sont définis entre les paires de valeurs scalaire/scalaire, vecteur/scalaire et vecteur/vecteur.

par exemple, prometheus_http_requests_total ***** 2

Il s'agit d'un exemple valide d'opérateurs arithmétiques entre le type de données vectorielles et le type de données scalaire où prometheus_http_requests_total est de type vectoriel et 2 est un scalaire.

Opérateurs binaires de comparaison Prometheus

Un opérateur de comparaison est un symbole mathématique utilisé pour la comparaison

Les opérateurs de comparaison binaire suivants existent dans Prometheus :

- == (égal)
- != (non égal)
- > (supérieur à)
- < (Inférieur à)
- >= (supérieur ou égal)
- <= (inférieur ou égal)

Les opérateurs de comparaison sont définis entre les paires de valeurs scalaire/scalaire, vecteur/scalaire et vecteur/vecteur.

par exemple, node_cpu_seconds_total > 300

Opérateurs binaires logiques/définis de Prometheus :

Les opérateurs logiques sont utilisés pour combiner des expressions relationnelles simples en expressions plus complexes. C'est définit seulement avec le vecteur instantané

Ces opérateurs binaires logiques/ensembles ne sont définis qu'entre vecteurs instantanés :

- and (intersection)
- or (syndicat)
- unless (à moins que , complément)

par exemple, vector 1 **and** vector 2, vector 1 **or** vector 2, vector 1 unless vector 2

prometheus_http_request_total or node_cpu_seconds_total

Prometheus Aggregation Operators

Les opérateurs d'agrégation calculent des valeurs mathématiques sur une plage de temps.

Types d'opérateurs d'agrégation :

- **sum** (calculer la somme sur les dimensions)
- **min** (sélectionnez le minimum sur les dimensions)
- **max** (sélectionnez le maximum sur les dimensions)
- **avg** (calculer la moyenne sur les dimensions)
- **group** (toutes les valeurs dans le vecteur résultant sont 1)
- **stddev** (calculer l'écart type de la population sur les dimensions)
- **stdvar** (calculer la variance standard de la population sur les dimensions)
- **count** (compter le nombre d'éléments dans le vecteur)
- **count_values** (compter le nombre d'éléments avec la même valeur)
- **bottomk** (le plus petit des k éléments par valeur d'échantillon)
- **topk** (k plus grands éléments par valeur d'échantillon)
- **quantile** (calculer le φ-quantile (0 ≤ φ ≤ 1) sur les dimensions)

Exemples

```
sum(node_cpu_seconds_total)
```

topk elements renvoie les k plus grands éléments de la série chronologique, donc exécutez-les en dessous de l'expression

```
topk(3,sum(node_cpu_seconds_total) by (mode))
```

Vous pouvez donc voir ici le top 3 des modes que votre CPU passe la plupart du temps

Les éléments **bottomk** renvoient les k éléments les plus petits de la série chronologique, alors exécutez-les sous l'expression

```
bottomk(3,sum(node_cpu_seconds_total) by (mode))
```

Vous pouvez donc voir ici le top 3 des modes que votre CPU passe moins de temps

Correspondance De Vecteur

Un par un

Les vecteurs sont égaux si les étiquettes sont égales.

Les API 5xx Représentent 10% Des Requêtes HTTP

```
rate(prometheus_http_requests_total{code=~"5.*"}[5m]) > .1  *
rate(prometheus_http_requests_total[5m])
```

Nous cherchons à représenter graphiquement chaque fois que plus de *10%* des requêtes HTTP d'une instance sont des erreurs. Avant de comparer les taux, PromQL vérifie d'abord que les étiquettes du vecteur sont égales.

Vous pouvez utiliser on pour comparer en utilisant certaines étiquettes ou ignoring pour comparer sur toutes les étiquettes sauf.

Plusieurs À Un

Il est possible d'utiliser des opérations de comparaison et d'arithmétique où un élément d'un côté peut être mis en correspondance avec de nombreux éléments de l'autre côté. *Vous devez explicitement dire à Prometheus quoi faire avec les dimensions supplémentaires.*

Vous pouvez utiliser group_left si le côté gauche a une cardinalité plus élevée, sinon utiliser group_right.

Exemples

Utilisation Du Processeur Par Instance

```
100 * (1 - avg by(instance)(irate(node_cpu_seconds_total{mode='idle'}[5m])))
```

Utilisation moyenne du processeur par instance pour une fenêtre de 5 minutes.

Utilisation De La Mémoire

```
node\_memory\_Active / on (instance) node\_memory\_MemTotal
```

Pourcentage de mémoire utilisé par l'instance.

Espace Disque

```
node_filesystem_avail_bytes{fstype!~"tmpfs|fuse.lxcfs|squashfs"} /
node_filesystem_size_bytes{fstype!~"tmpfs|fuse.lxcfs|squashfs"}
```

Pourcentage d'espace disque utilisé par l'instance. Nous sommes à la recherche de l'espace disponible, sans tenir compte des cas qui ont tmpfs, fuse.lxcfsou squashfsdans leur fstypeet divisant par leur taille totale.

Taux D'erreur HTTP En% Du Trafic

```
rate(prometheus_http_requests_total{code=~"5.*"}[5m]) /
rate(prometheus_http_requests_total[5m])
```

Alertes Déclenchées Au Cours Des Dernières 24 Heures

```
sum(sort_desc(sum_over_time(ALERTS{alertstate="firing"}[24h]))) by (alertname)
```

Vous pouvez trouver des exemples plus utiles https://github.com/infinityworks/prometheus-example-queries

Exemples supplémentaires

Vous pouvez agréger les valeurs métriques par dimensions arbitraires en utilisant «by» ou «without»:

```
sum(node_scrape_collector_duration_seconds) without (collector)
```

Ces opérateurs d'agrégation sont familiers si vous connaissez déjà SQL. PromQL a également `min()`, `max()`, `avg()`, `count()` et [plus](https://prometheus.io/docs/prometheus/latest/querying/operators/# opérateurs d'agrégation).
Exercice: écrivez une requête qui retourne les 5 collecteurs prenant le plus de temps lors des grattage scrapping.

Solution

```
topk(5, node_scrape_collector_duration_seconds)
```

PromQL prend également en charge la correspondance vectorielle (vector match) pour les opérations binaires et arithméthiques. Allons-y générer des requêtes invalides à Prometheus:

```
for i in {1..10}; do \
  curl localhost:9090/api/v1/query; \
  curl localhost:9090/static/notfound; \
  sleep 5
done
```

`http_requests_total` renvoie le nombre de requêtes reçue par Prometheus Calculons le total des requêtes HTTP reçues ayant renvoyé le code 400.

```
prometheus_http_requests_total{code="400"}
```

Ensuite, pour obtenir le pourcentage de demandes réussies, nous pouvons utiliser l'opérateur de division pour obtenir le nombre total de réponses 400 avec le nombre total de réponses et soustraire de 1

```
1 -
(sum(prometheus_http_requests_total{code="400"})
  /
 sum(prometheus_http_requests_total))
```

Et maintenant, nous pouvons demander à Prometheus le pourcentage de requêtes HTTP qui ont renvoyé un code d'état 400.

```
100 *
  sum(prometheus_http_requests_total{code="400"})
   /
  sum(prometheus_http_requests_total)
```

Les requêtes ci-dessus nous donnent le taux d'erreur pour le dernier instant de temps pour lequel nous avons une mesure, mais pas pour une plage de temps donnée. Rappelez-vous que nous pouvons interroger une fenêtre de temps en utilisant un sélecteur de plage, c'est-à- dire [5m], essayons cela ici:

```
100 * sum(rate(prometheus_http_requests_total{code="400"}[5m])) /
sum(rate(prometheus_http_requests_total[5m]))
```

***Exercice**: modifiez la requête pour calculer le pourcentage de requêtes HTTP ayant renvoyé un code d'état entre 400 et 499.*

Solution

```
100 * sum(rate(prometheus_http_requests_total{code=~"4.."}[5m])) /
sum(rate(prometheus_http_requests_total[5m]))
```

Calculons maintenant les quantiles. La méthode dépend du fait que la métrique est un summary ou un histogramme. Les summary peuvent être reconnus par leur étiquette **`quantile`** tandis que les métriques d'histogramme ont une étiquette **`le`** qui représente le compartiment de l'histogramme (le = less or equal). Voici un exemple de métrique récapitulative.

```
go_gc_duration_seconds
```

Prometheus expose des métriques d'histogramme mesurant la taille des réponses HTTP.

```
prometheus_http_response_size_bytes_bucket{handler="/api/v1/query"}
```

La fonction **`histogram_quantile()`** appliquée aux métriques d'histogramme renvoie une estimation du n-ième quantile.

```
histogram_quantile(0.9,
rate(prometheus_http_response_size_bytes_bucket{handler="/api/v1/query"}[5m]))
```

Les summary et les histogrammes suivent également la somme et le nombre d'échantillons observés qui peuvent être utilisés pour calculer les valeurs moyennes:

```
sum(rate(prometheus_http_request_duration_seconds_sum[5m])) by (job, handler)
 /
sum(rate(prometheus_http_request_duration_seconds_count[5m])) by (job, handler)
```

Exercice:

Excluez les valeurs NaN.

Solution

```
sum(rate(prometheus_http_request_duration_seconds_sum[5m])) by (job, handler)
 /
( sum(rate(prometheus_http_request_duration_seconds_count[5m])) by (job, handler) > 0 )
```

Exercices d'applications

1. Triez la taille libre de votre système de fichier en ordre croissant puis décroissant

```
sort (node_filesystem_free_bytes)
sort_desc (node_filesystem_free_bytes)
```

2. Affichez les 2 partitions les plus élévées (enterme de stockage) dans le systèmes du fichier pour l'instance « prometheus »

```
topk (2, node_filesystem_free_bytes{ instance="localhost:9100"})
```

3. Affichez la taille totale de chaque système de fichiers, avec les mêmes étiquettes qu'avant

```
node_filesystem_free_bytes / node_filesystem_size_bytes
```

4. Déterminez le nombre de système de fichiers (nombre des partitions) disponible sur chaque instance

```
count by (instance) (node_filesystem_size_bytes)
```

5. Déterminer le nombre total de réponses http du serveur prometheus pour l'ensemble des requêtes http erronés

```
sum(prometheus_http_requests_total{code="400"})
```

6. Déterminer le pourcentage de réponses http du serveur prometheus pour l'ensemble des requêtes http réussis.

```
1-(sum(prometheus_http_requests_total{code="400"}) / sum(prometheus_http_requests_total))
```

7. Déterminer le pourcentage de réponses http du serveur prometheus pour l'ensemble des requêtes http réussis pendant l'intervalle de temps de 15 min avant une heure.

Une réponse directe est :

```
1 - (sum(prometheus_http_requests_total{code="400"}[30s]) /
sum(prometheus_http_requests_total[30s]))
```

mais celle la va afficher un message d'erreure

Error executing query: invalid parameter "query": 1:8: parse error: expected type instant vector in aggregation expression, got range vector

⇨ Donc il faut transformer le rang vector en instant vector en passant par la fonction rate

```
1-(sum(rate(prometheus_http_requests_total{code="400"}[5m])) /
sum(rate(prometheus_http_requests_total[5m])))
```

Puis pour ajouter l'offset

```
1-(sum(rate(prometheus_http_requests_total{code="400"}[15m] offset 1h)) /
sum(rate(prometheus_http_requests_total[15m] offset 1h)))
```